# O Metodă de Coborâre Duală a Coordonatelor pentru SVM Liniar de Dimensiuni Mari – Prezentare Generală

*Lucrare Originală Scrisă de Cho-Jui Hsieh, Kai-Wei Chang, Chih-Jen Lin, S. Sathiya Keerthi, S. Sundararajan*

***Proiect Realizat de Dumitrache Larisa și Stegărescu Ana-Maria***

*~ Mai 2021 ~*

## ~ Rezumat ~

Mașiniile cu suport vectorial liniar reprezintă una dintre cele mai populare și des folosite instrumente pentru clasificarea unui volum ridicat de date de intrare. Lucrarea prezentată discută o metodă nouă de **SVM liniar**, simplă și care ajunge la o soluție cu acuratețea $\epsilon$ în $O(\log(1/\epsilon))$ iterații. Experimentele realizate indică faptul că acest procedeu este mult mai rapid și eficient decât metodele consacrate de la momentul scrierii lucrării (cum ar fi *Pegasos, TRON, SVM*$^{perf}$ etc...).

# Cuprins

# 1. Introducere

## *1.1. Prefață*

În multe situații, dorim ca algoritmul nostru de învățare automată să prezică unul dintre rezultatele posibile. Spre exemplu, un utilizator dorește să sorteze mail-urile primite în e-mail personal și spam, deci avem 2 categorii posibile. Un alt exemplu este cel al unui telescop care identifică dacă un corp ceresc este o galaxie, o stea sau o planetă; în acest caz, rezultând 3 clase de obiecte. După cum se poate remarca și din cazurile anterioare, în practică, există un număr redus de rezultate pe care trebuie să le obținem și, mai important, nu avem structuri suplimentare derivate din aceste rezultate principale. În cele ce urmează, vom considera doar cazurile care au outcome-uri binare, adică sunt generate doar 2 rezultate posibile. Pentru acest tip de clasificare, cea mai cunoscută abordare este SVM-ul, care constituie și subiectul lucrării analizate.

## *1.2. Cunoștințe Teoretice Inițiale*

Mașinile cu suport vectorial, SVM pe scurt (eng. original „*Support Vector Machines*") au fost introduse în 1992 și au devenit unul dintre cele mai utilizate și prolifice instrumente de clasificare a datelor.

Având un set de perechi instanță-etichetă $(x_i, y_i), i = 1, \dots, l; x_i \in \mathbb{R}^n, y_i \in \{-1, +1\}$, SVM rezolvă următoarea problemă de optimizare fără restricții, numită și **forma primară**:

$$\min_{\boldsymbol{w}} \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C\sum_{i=1}^{l}\xi(\boldsymbol{w}, x_i, y_i) \qquad (1)$$

unde $\xi(\boldsymbol{w}, x_i, y_i)$ este funcția de pierdere și $C > 0$ este un parametru de penalizare. Două funcții de pierdere des întâlnite sunt:

- L1-SVM: $max(1 - y_i\boldsymbol{w}^T x_i, 0)$
- L2-SVM: $max(1 - y_i\boldsymbol{w}^T x_i, 0)^2$ (2)

În unele aplicații, o problemă de SVM apare cu un bias $b$, iar, în aceste cazuri, adăugăm o dimensiune suplimentară, astfel: $x_i{}^T \leftarrow [x_i{}^T, 1]$ $\boldsymbol{w}^T \leftarrow [\boldsymbol{w}^T, b]$ (3)

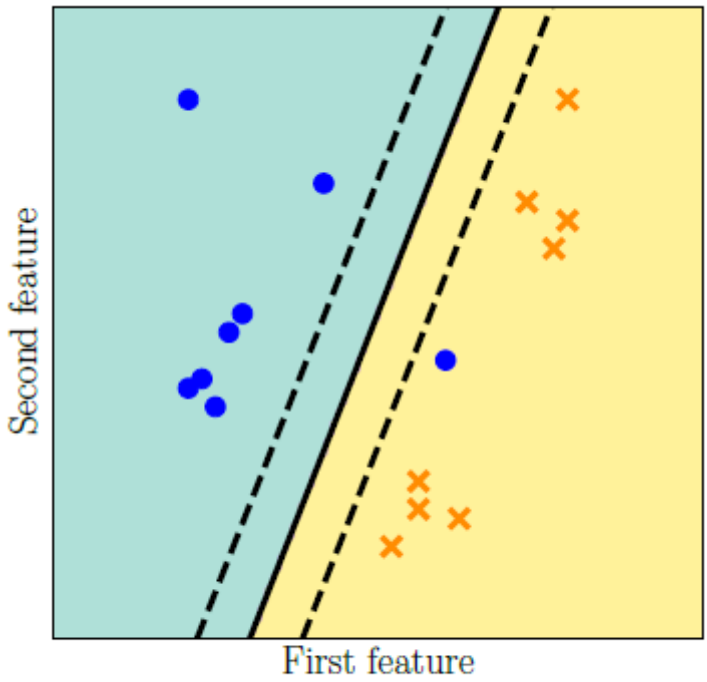
În loc de forma primară, putem rezolva **problema duală**:

$$\min_{\boldsymbol{\alpha}} f(\boldsymbol{\alpha}) = \frac{1}{2}\boldsymbol{\alpha}^T\bar{Q}\boldsymbol{\alpha} - e^T\boldsymbol{\alpha}, \text{ constrâns după } 0 \le \alpha_i \le U, \forall i \qquad (4)$$

unde $\bar{Q} = Q + D, D$ este o matrice diagonală și $Q_{ij} = y_i y_j x_i{}^T x_j$. Pentru **L1-SVM**, $U = C$ și $D_{ii} = 0, \forall i$; iar pentru **L2-SVM**, $U = \infty$ și $D_{ii} = 1/(2C), \forall i$.

Un SVM transformă vectorii de antrenat neliniari în vectori liniari folosind o funcție neliniară $\phi(\boldsymbol{x})$. Datorită dimensiunii mari a variabilei vectoriale $\boldsymbol{w}$, problema (4) se rezolvă prin trucul kernel-ului, adică se folosește o formă închisă pentru $\phi(\boldsymbol{x_i})^T\phi(\boldsymbol{x_j})$. Numim o astfel de problemă drept **SVM neliniar**. În unele aplicații, datele de intrare sunt într-un număr atât de

ridicat și prezintă atât de multe caracteristici încât performanțele sunt similare chiar dacă folosim sau nu maparea nonliniară. Dacă datele nu sunt mapate, în cele mai multe cazuri, putem antrena seturi de date mult mai mari. Numim aceste cazuri drept **SVM liniar** și se întâlnesc, adesea, la clasificarea documentelor. Lucrarea abordată își propune să rezolve ***probleme SVM liniare foarte mari***.




(a) SVM with linear kernel

## *1.3. Inspirație*

Au fost propuse multe metode pentru problemele SVM liniare cu un set mare de date. Chang și alți cercetători au recomandat, în anul 2008, utilizarea metodei de coborâre a coordonatelor pentru a rezolva *forma primară L2-SVM*. Experimentele au arătat că această abordare obține mult mai rapid un model funcționabil decât propunerile anterioare. Coborârea coordonatelor este o tehnică populară de optimizare și constă în actualizarea unei variabile prin minimizarea unei subprobleme cu o singură variabilă. Datorită nediferențiabilității formei primare L1-SVM, Chang și echipa sa s-au limitat la L2-SVM. În plus, datorită faptului că forma primară L2-SVM este diferențiabilă, dar nu dublu-diferențiabilă, anumite considerații sunt necesare în rezolvarea subproblemei.

În schimb, funcția obiectiv a formei duale (4) este dublu-diferențiabilă, atât pentru L1-SVM, cât și pentru L2-SVM. Există mai multe lucrări care au discutat metode de coborâre a coordonatelor pentru problema duală (4), dar nu s-au concentrat pe date de intrare cu dimensiuni mari folosind kernel-ul liniar. Textul de bază analizează aceste metode, dar cu evidente îmbunătățiri: adică este tratată problema din perspectiva unui set de date foarte stufos și, de asemenea, se profită de kernel-ul liniar care avantajează aceaste metode, ducând la performanțe ridicate (față de alte abordări) atunci când numărul de date și caracteristici este voluminos.

# 2. Metoda de Coborâre a Coordonatelor Propusă

## *2.1. Idee Generală*

Lucrarea de față demonstrează că o soluție optimă este obținută în $\boldsymbol{O(\log(1/\epsilon))}$ iterații. După cum am discutat în capitolul anterior, se deduce ușor că alegerea subproblemei cu o singură variabilă determină complexitatea și este de interes pentru eficientizarea problemei de optimizare. Autorii propun o implementare în care este folosită o ordina aleatoare a subproblemelor, la fiecare iterație, ceea ce duce la o antrenare promptă. Experimentele au indicat că această metodă este mult mai eficientă decât rezolvarea formei primare. Chang și echipa lui s-au concentrat pe forma primară de SVM, folosind valorile de date corespunzătoare caracteristicilor. Cu toate acestea, în practică, s-a dovedit că se accesează mai ușor valorile propriu-zise. Algoritmul propus de lucrare folosește acest avantaj, astfel încât implementarea devine mai simplă decât cea a lui Chang (2008).

## *2.2. Descrierea Metodei*

În această secțiune, vom descrie metoda de coborâre duală a coordonatelor pentru L1-SVM și L2-SVM.

Procesul de optimizare pornește de la o valoare inițială $\alpha^0 \in \mathbb{R}^l$ și se generează o secvență de vectori $\{\alpha^k\}_{k=0}^{\infty}$. Numim ***iterație externă*** procesul de trecere de la $\alpha^k$ la $\alpha^{k+1}$. În fiecare iterație externă, avem $l$ ***iterații interne*** astfel încât $\alpha_1, \alpha_2, \dots, \alpha_l$ să fie actualizate secvențial. Deci, fiecare iterație externă generează vectori $\alpha^{k,i} \in \mathbb{R}^l, i = 1, \dots, l+1$ astfel încât:

$$\begin{cases} \alpha^{k,1} = \alpha^k \\ \alpha^{k,l+1} = \alpha^{k+1} \\ \alpha^{k,i} = [\alpha_1^{k+1}, \dots, \alpha_{i-1}^{k+1}, \alpha_1^k, \dots, \alpha_l^k]^T, \forall i = 2, \dots l \end{cases}$$

Pentru a trece de la $\alpha^{k,i}$ la $\alpha^{k,i+1}$, rezolvăm următoarea subproblemă cu o singură variabilă: $\min_d f(\boldsymbol{\alpha^{k,i}} + d\boldsymbol{e_i})$, cu $0 \le \alpha_i^k + d \le U$ (5), unde $\boldsymbol{e_i} = [0, \dots, 0, 1, 0, \dots, 0]^T$. Funcția obiectiv pentru (5) este o simplă funcție pătratică după $d$:

$$f(\boldsymbol{\alpha^{k,i}} + d\boldsymbol{e_i}) = \frac{1}{2}\overline{Q_{ii}}d^2 + \nabla_i f(\boldsymbol{\alpha^{k,i}})d + k \qquad (6),$$

unde $\nabla_i f$ este al $i$-lea element al gradientului $\nabla f$ și $k$ este o constantă. Se poate remarca cu ușurință că (5) are un optim pentru $d = 0$ (adică nu trebuie să actualizăm $\alpha_i$) ddacă

$$\nabla_i^P f(\boldsymbol{\alpha}^{k,i}) = 0 \qquad (7),$$

unde $\nabla_i^P f(\alpha)(gradientul\ proiectat) = \begin{cases} \nabla_i f(\alpha), & pentru\ 0 < \alpha_i < U \\ \min(0, \nabla_i f(\alpha)), & pentru\ \alpha_i = 0 \\ \max(0, \nabla_i f(\alpha)), & pentru\ \alpha_i = U \end{cases}$ (8)

Dacă (7) este adevărat, trecem la iterația $i+1$, fără să actualizăm $\alpha_i^{k,i}$. Altfel, trebuie să găsim soluția optimă pentru (5). Dacă $\overline{Q_{ii}} > 0$, soluția este:

$$\alpha_i^{k,i+1} = \min\left(max\left(\alpha_i^{k,i} - \frac{\nabla_i f\left(\alpha^{k,i}\right)}{\overline{Q_{ii}}}, 0\right), U\right) \qquad (9)$$

Deci, trebuie să calculăm $\overline{Q_{ii}}$ și $\nabla_i f\left(\alpha^{k,i}\right)$. În primul rând, $\overline{Q_{ii}} = x_i^T x_i + D_{ii}$ poate fi precalculat și reținut în memorie. În al doilea rând, pentru a evalua $\nabla_i f\left(\alpha^{k,i}\right)$ folosim:

$$\nabla_i f(\alpha) = (\bar{Q}\alpha)_i - 1 = \sum_{j=1}^{l} \overline{Q_{ij}}\alpha_j - 1 \qquad (10)$$

Este posibil ca $\bar{Q}$ să fie prea mare pentru a fi reținut în memorie, de aceea trebuie să calculăm al $i$-lea rând a lui $\bar{Q}$ când rezolvăm (10). Dacă $\bar{n}$ este numărul mediu de elemente nenule pe iterație și $O(\bar{n})$ este complexitatea necesară pentru fiecare evaluare de kernel, atunci, pentru a obține al $i$-lea rând al matricei kernel, vom avea complexitatea $O(l\bar{n})$. Astfel de operații sunt costisitoare din punct de vedere al timpului de execuție. Dar, pentru un SVM liniar, putem defini:

$$w = \sum_{j=1}^{l} y_j\, \alpha_j x_j \qquad (11),$$

deci (10) devine $$\nabla_i f(\alpha) = y_i w^T x_i - 1 + D_{ii}\alpha_i \qquad (12).$$

Pentru a calcula $w^T x_i$ avem o complexitate de $O(\bar{n})$, care este mult mai mică decât $O(l\bar{n})$. Ca să putem aplica (12), trebuie să menținem $w$ pe tot parcursul procedurii. Ca să calculăm $w$ după formula (11) avem nevoie de $O(l\bar{n})$ operații, deci nu obținem o complexitate prea mare. Din fericire, dacă $\bar{\alpha_i}$ este valoarea curentă și $\alpha_i$ este valoarea după actualizare, putem menține $w$ folosind formula: $$w \leftarrow w + (\alpha_i - \bar{\alpha_i})y_i x_i \qquad (13).$$

În acest caz, numărul de operații este egal doar cu $O(\bar{n})$. Valoarea inițială a lui $w$ este 0, deoarece $\alpha^0 = 0$. În final, obținem valoarea optimă a lui $w$ pentru forma primară (1), deoarece relația dintre forma primară și cea duală implică formula (11).

Dacă $\overline{Q_{ii}} = 0 \Rightarrow D_{ii} = 0,\ Q_{ii} = {x_i}^T x_i = 0$, deci $x_i = 0$. Această egalitate are loc doar pentru L1-SVM, fără bias (formula (3)). Din (12), dacă $x_i = 0$, atunci $\nabla_i f\left(\alpha^{k,i}\right) = -1$. Cum $U = C < \infty$ pentru L1-SVM, conform lui (5), $\alpha_i^{k,i+1} = U$. Putem include acest caz în (9) dacă setăm $1/\overline{Q_{ii}} = \infty$.

## 2.2.1. Algoritmul 1

**Algorithm 1** A dual coordinate descent method for Linear SVM

- Given $\boldsymbol{\alpha}$ and the corresponding $\boldsymbol{w} = \sum_i y_i \alpha_i \boldsymbol{x}_i$.
- While $\boldsymbol{\alpha}$ is not optimal

  For $i = 1, \ldots, l$

  (a) $G = y_i \boldsymbol{w}^T \boldsymbol{x}_i - 1 + D_{ii}\alpha_i$

  (b)

$$PG = \begin{cases} \min(G, 0) & \text{if } \alpha_i = 0, \\ \max(G, 0) & \text{if } \alpha_i = U, \\ G & \text{if } 0 < \alpha_i < U \end{cases}$$

  (c) If $|PG| \neq 0$,

$$\bar{\alpha}_i \leftarrow \alpha_i$$
$$\alpha_i \leftarrow \min(\max(\alpha_i - G/\bar{Q}_{ii}, 0), U)$$
$$\boldsymbol{w} \leftarrow \boldsymbol{w} + (\alpha_i - \bar{\alpha}_i) y_i \boldsymbol{x}_i$$

## *2.3. Pe Scurt*

Algoritmul prezentat folosește (12) pentru a calcula $\nabla_i f(\alpha^{k,i})$, verifică optimalitatea subproblemei (5) folosind condiția (7), actualizează $\alpha_i$ după relația (9) și menține $w$ cu ajutorul lui (13). Complexitatea pe iterație (adică de la $\alpha^k$ la $\alpha^{k+1}$) este $O(l\bar{n})$. Memoria este folosită pentru a reține $x_1, \ldots, x_l$.

Pentru convergență, autorii definesc următoarea teoremă, care este demonstrată folosind tehnicile din Luo & Tseng (1992):

> ***Teorema 1:*** Pentru L1-SVM și L2-SVM, $\{\alpha^{k,i}\}$ generat de algoritmul 1 converge global la o soluție optimală $\alpha^*$. Rata de **convergență este cel puțin liniară**: avem $0 < \mu < 1$ și o iterație $k_0$ astfel încât $f(\alpha^{k+1}) - f(\alpha^*) \leq \mu\left(f(\alpha^k) - f(\alpha^*)\right), \forall\, k \geq k_0$ (14)

Rezultatul convergenței este remarcabil. De obicei, pentru o problemă convexă, dar nu strict-convexă (ex: L1-SVM), se obține doar că orice punct limită este optim. Autorii au definit o soluție $\alpha$ de acuratețe $\epsilon$ dacă $f(\alpha) \leq f(\alpha^*) + \epsilon$. Din (14), se observă ușor că algoritmul definit anterior obține o soluție cu acuratețea $\epsilon$ în $O(\log(1/\epsilon))$ iterații.

# 3. Aspecte ale Implementării

## 3.1. Permutarea Aleatoare a Subproblemelor

Algoritmul 1 rezolvă subproblemele în ordinea originală, i.e. $\alpha_1, \dots, \alpha_l$. Rezultatele anterioare au demonstrat faptul că se obține o convergență mai rapidă dacă subproblemele cu o singură variabilă sunt abordate într-o ordine aleatoare. Matematic, la fiecare iterație externă, se permută mulțimea $\{1, \dots, l\}$ și se obține $\{\pi(1), \dots, \pi(l)\}$, iar subproblemele sunt rezolvate în ordinea dată de această nouă permutare (notație: $\alpha_{\pi(i)}\ pentru\ i = 1, \dots, l$ ). Similar cu algoritmul 1, este generată o secvență $\{\alpha^{k,i}\}$ astfel încât $\alpha^{k,1} = \alpha^k$, $\alpha^{k,l+1} = \alpha^{k+1,1}$ și

$$\alpha_t^{k,i} = \begin{cases} \alpha_t^{k+1};\ pentru\ \pi_k^{-1}(t) < i \\ \alpha_t^k;\ \ pentru\ \pi_k^{-1}(t) \geq i \end{cases}$$

Actualizarea lui $\alpha^{k,i}$ la $\alpha^{k,i+1}$ se realizează astfel:

$$\alpha_t^{k,i+1} = \alpha_t^{k,i} + \arg \min_{0 \leq \alpha_t^{k,i} + d \leq U} f(\boldsymbol{\alpha}^{k,i} + d\boldsymbol{e}_t), pentru\ \pi_k^{-1}(t) = i$$

Teorema 1 este validă și pentru acest model, deci se obține o soluție cu acuratețea $\epsilon$ în $O(\log(1/\epsilon))$ iterații. Prin experimente, se remarcă că această îmbunătățire scade timpul de execuție față de algoritmul 1.

## 3.2. În Cadrul Practic

Cadrul practic face referire la cel descris de Collins et al., în 2008, care vizează problemele de clasificare cu mai multe clase ( > 2 ). Lucrarea de față se focusează doar pe clasificările binare, dar, în unele aplicații, numărul de pași este foarte mare, deci a trece prin toate subproblemele $\alpha_1, \dots, \alpha_l$ provoacă o iterație exterioară costisitoare din punct de vedere al timpului și memoriei. În schimb, se poate alege un index random $i_k$, la un moment dat, apoi se actualizează doar $\alpha_{ik}$ la a $k$-a iterație exterioară; obținând un algoritm similar cu ideile prezentate în sursa de inspirație menționată mai sus.

### 3.2.1. Algoritmul 2

**Algorithm 2** Coordinate descent algorithm with randomly selecting one instance at a time

- Given $\boldsymbol{\alpha}$ and the corresponding $\boldsymbol{w} = \sum_i y_i \alpha_i \boldsymbol{x}_i$.
- While $\boldsymbol{\alpha}$ is not optimal
  - Randomly choose $i \in \{1, \dots, l\}$.
  - Do steps (a)-(c) of Algorithm 1 to update $\alpha_i$.

Conform demonstrației prezentate în lucrare, acest algoritm este puternic convex.

## 3.3. Micșorarea Problemei de Optimizare

După cum am discutat de la început, lucrarea analizată face referire doar la SVM-ul de tip liniar. Experimentele au demonstrat că metoda de coborâre a coordonatelor folosind micșorarea dimensiunii problemei de optimizare este mult mai eficientă din punct de vedere al timpului decât metodele de descompunere, după cum este exemplificat în tabelul următor:

*Table 1.* A comparison between decomposition methods (Decomp.) and dual coordinate descent (DCD). For both methods, we consider that one $\alpha_i$ is updated at a time. We assume Decomp. maintains gradients, but DCD does not. The average number of nonzeros per instance is $\bar{n}$.

| | Nonlinear SVM | | Linear SVM | |
|---|---|---|---|---|
| | Decomp. | DCD | Decomp. | DCD |
| Update $\alpha_i$ | $O(1)$ | $O(l\bar{n})$ | $O(1)$ | $O(\bar{n})$ |
| Maintain $\nabla f(\boldsymbol{\alpha})$ | $O(l\bar{n})$ | NA | $O(l\bar{n})$ | NA |

În ce constă această micșorare? Este vorba de algoritmul implementat în biblioteca LIBSVM, creată de Chang & Lin, în 2011. Pe scurt, anumite elemente sunt eliminate din setul original, rezultând astfel într-o problemă de dimensiuni mai mici.

### 3.3.1. Algoritmul 3

- Given $\epsilon, \boldsymbol{\alpha}$ and the corresponding $\boldsymbol{w} = \sum_i y_i \alpha_i \boldsymbol{x}_i$.
- Let $\bar{M} \leftarrow \infty, \bar{m} \leftarrow -\infty$.
- Let $A \leftarrow \{1, \ldots, l\}$.
- While
  1. Let $M \leftarrow -\infty, m \leftarrow \infty$.
  2. For all $i \in A$
     (a) $G = y_i \boldsymbol{w}^T \boldsymbol{x}_i - 1 + D_{ii}\alpha_i$
     (b) $PG \leftarrow 0$
         If $\alpha_i = 0$,
           $A \leftarrow A \backslash \{i\}$ and CONTINUE if $G > \bar{M}$
           $PG \leftarrow G$ if $G < 0$
         Else If $\alpha_i = U$,
           $A \leftarrow A \backslash \{i\}$ and CONTINUE if $G < \bar{m}$
           $PG \leftarrow G$ if $G > 0$
         Else
           $PG \leftarrow G$
     (c) $M \leftarrow \max(M, PG), m \leftarrow \min(m, PG)$
     (d) If $|PG| \neq 0$,
           $\bar{\alpha}_i \leftarrow \alpha_i$
           $\alpha_i \leftarrow \min(\max(\alpha_i - G/\bar{Q}_{ii}, 0), U)$
           $\boldsymbol{w} \leftarrow \boldsymbol{w} + (\alpha_i - \bar{\alpha}_i) y_i \boldsymbol{x}_i$
  3. If $M - m < \epsilon$
     (a) If $A = \{1, \ldots, l\}$,
           BREAK.
         Else,
           $A \leftarrow \{1, \ldots, l\}, \bar{M} \leftarrow \infty, \bar{m} \leftarrow -\infty$.
           (i.e., no shrinking at the next iteration)
  4. If $M \leq 0$, then $\bar{M} \leftarrow \infty$. Else $\bar{M} \leftarrow M$.
  5. If $m \geq 0$, then $\bar{m} \leftarrow -\infty$. Else $\bar{m} \leftarrow m$.

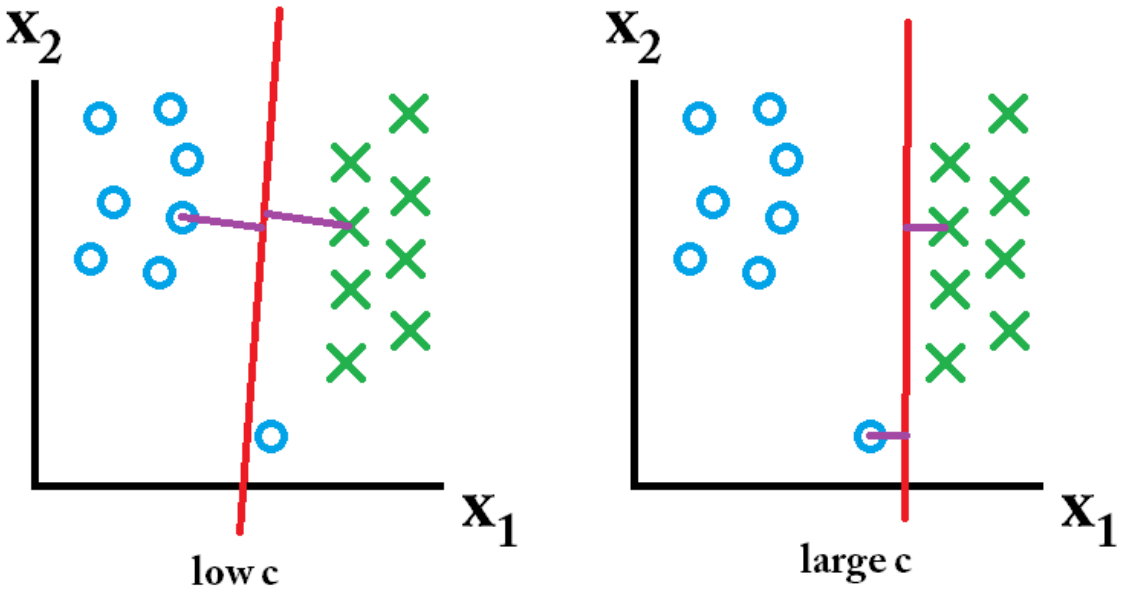
# 4. Contribuția Noastră

## *4.1. Explicații Introductive și Analiza Algoritmilor*

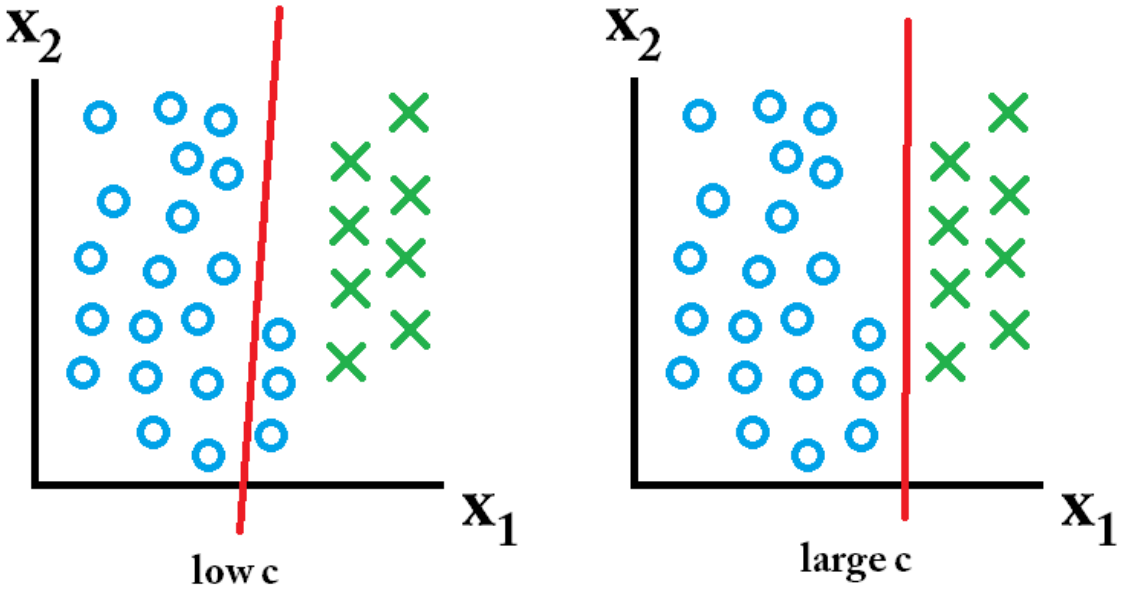
Algoritmii care au fost aleși pentru a fi analizați și implementați sunt cei 3, prezentați în lucrarea originală la paginile 3, 4, respectiv 11 și care au fost menționați mai sus.

Pentru a putea să îi analizăm, trebuie să înțelegem, măcar conceptual, ceea ce se întâmplă în interiorul algoritmilor, care sunt parametrii de input, de output și ce semnifică fiecare notație menționată de autori.

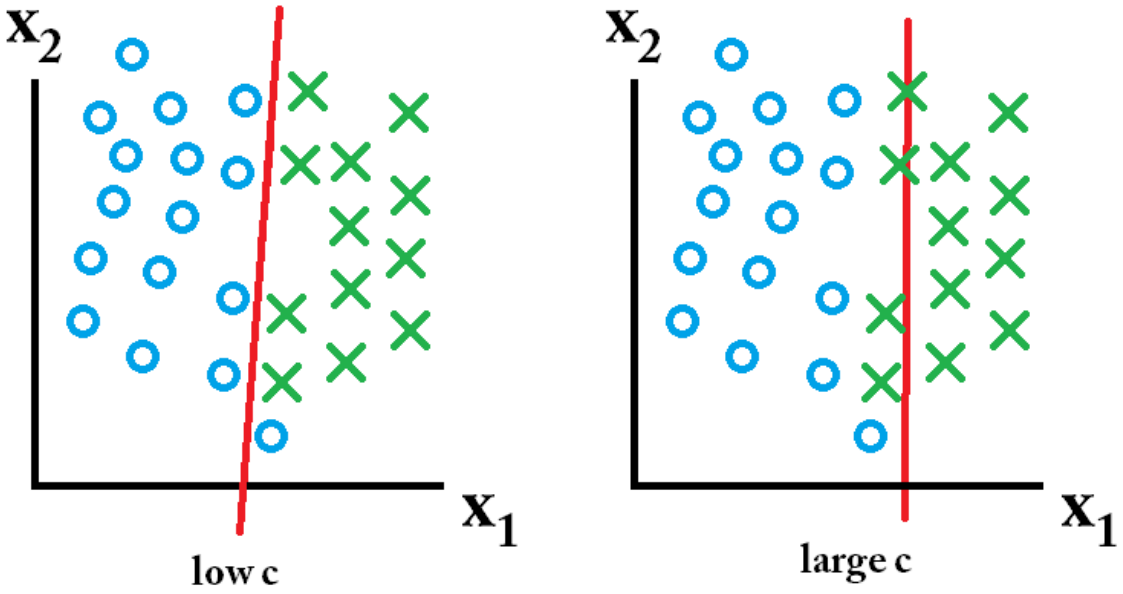
După cum știm, SVM este folosit pentru a clasifica dateset-uri. Dar ce este acel *C*, folosit în acești algoritmi și la ce ajută el? După cum remarcăm din formula (4), *C* este foarte important, pentru că în funcție de el aflăm *U*, *D* și *Q*. Numele ne spune că este ***parametrul de penalizare*** și vedem cu ochiul liber, din (1), că se leagă de funcția de pierdere. Pe scurt, parametrul *C* spune optimizării SVM cât de mult dorim să evităm clasificarea greșită a fiecărui exemplu de antrenament. Într-un SVM ne interesează 2 lucruri: un hiperplan cu cea mai mare margine minimă și un hiperplan care separă corect cât mai multe instanțe posibil. În practică, se întâmplă adesea să nu putem obține ambele condiții; astfel, prin intermediul lui *C* indicăm SVM-ului cât de precisă să fie separarea identităților din dataset. Spre exemplu, în imaginea de mai jos, primul




*C* obține o margine largă, dar neglijează acel unic cerculeț albastru; problema este redresată de al doilea *C*. Însă care dintre abordări este mai bună? Acest lucru depinde de aspectul datelor viitoare pe care le vom prezice și, cel mai adesea, nu le vom cunoaște, în prealabil. Dacă datele viitoare arată astfel:




... atunci clasificatorul cu valoarea *C* mare este cel mai bun. Dar dacă avem următorul caz:




... atunci clasificatorul care a folosit primul *C* este mult mai indicat. De aceea este important să alegem o valoare cât mai potrivită pentru acest parametru. Concluzie: Pentru valori mari de *C*, optimizarea va alege un hiperplan cu marjă mai mică dacă hiperplanul face o treabă mai bună de a clasifica corect toate punctele de antrenament. În schimb, o valoare foarte mică a lui *C* va face ca optimizatorul să caute un hiperplan de separare cu marjă mai mare, chiar dacă hiperplanul respectiv clasifică greșit mai multe puncte. Pentru valori foarte mici ale lui *C*, ar trebui să fie obținute exemple clasificate greșit foarte des, chiar dacă datele de antrenament sunt separabile liniar.

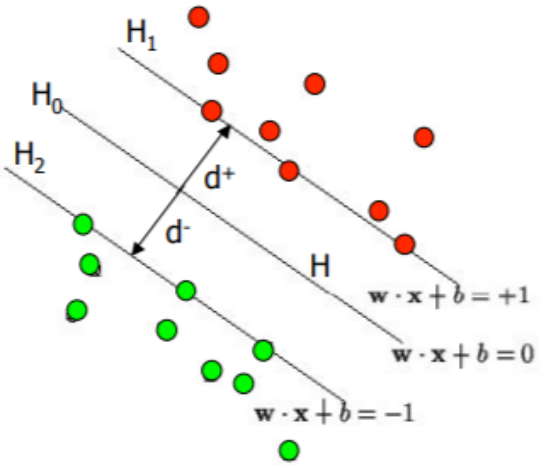
Acum că am lămurit acest aspect, să înțelegem ce este acel *w*, de ce trebuie să îl obținem din algoritm și care este utilizarea lui. *W* este vectorul de „greutate”. Cu ajutorul lui facem predicțiile. Mai jos avem câteva poze care prezintă, mai detaliat, relațiile matematice dintre w,

# Definitions

Define the hyperplanes $H$ such that:

$w \bullet x_i + b \geq +1$ when $y_i = +1$

$w \bullet x_i + b \leq -1$ when $y_i = -1$




$H_1$ and $H_2$ are the planes:

$H_1$: $w \bullet x_i + b = +1$

$H_2$: $w \bullet x_i + b = -1$

The points on the planes $H_1$ and $H_2$ are the tips of the Support Vectors

The plane $H_0$ is the median in between, where $w \bullet x_i + b = 0$

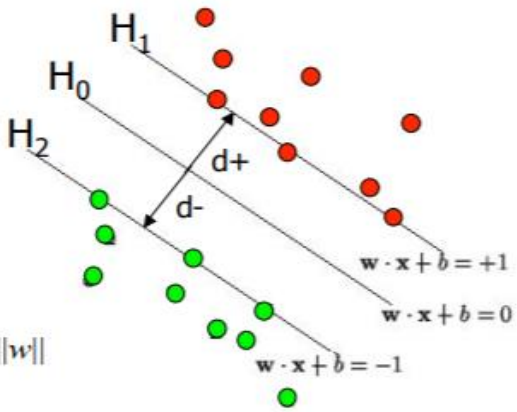
d+ = the shortest distance to the closest positive point

d- = the shortest distance to the closest negative point

The margin (gutter) of a separating hyperplane is d+ + d–.

# Maximizing the margin (aka street width)

We want a classifier (linear separator) with as big a margin as possible.




Recall the distance from a point$(x_0,y_0)$ to a line: $Ax+By+c = 0$ is: $|Ax_0 + By_0 + c|/\text{sqrt}(A^2+B^2)$, so,

The distance between $H_0$ and $H_1$ is then: $|w \bullet x+b|/||w|| = 1/||w||$, so

The total distance between $H_1$ and $H_2$ is thus: $2/||w||$

In order to maximize the margin, we thus need to minimize $||w||$. With the condition that there are no datapoints between $H_1$ and $H_2$:

$\mathbf{x}_i \bullet \mathbf{w} + b \geq +1$ when $y_i = +1$

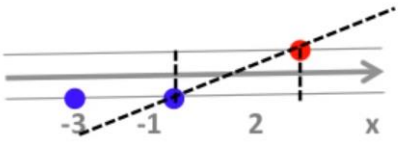
$\mathbf{x}_i \bullet \mathbf{w} + b \leq -1$ when $y_i = -1$

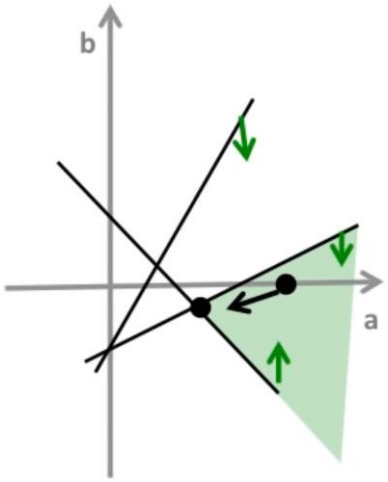
**Can be combined into: $y_i(\mathbf{x}_i \bullet \mathbf{w}) \geq 1$**

... datele de intrare și hiperplan. Un exemplu concret, pentru a înțelege cum se calculează *w*:

# A 1D Example

- Suppose we have three data points
  x = -3, y = -1
  x = -1, y = -1
  x = 1, y = 1
- Many separating perceptrons, T[ax+b]
  - Anything with ax+b = 0 between -1 and 2
- We can write the margin constraints
  a (-3) + b < -1 => b < 3a - 1
  a (-1) + b < -1 => b < a - 1
  a (2) + b > +1 => b > -2a + 1
- Ex: a = 1, b = 0
- Minimize ||a|| => a = .66, b = -.33
  - Two data on the margin; constraints "tight"




Acum că am înțeles notațiile de bază, să vedem care sunt diferențele dintre cei 3 algoritmi propuși. Atât algoritmul 2, cât și 3, sunt variațiuni ale algoritmului 1. La **2** se reduce considerabil numărul de pași deoarce, se renunță la acel „*for i = 1, ..., l*" de la pasul 2 și se înocuiește cu o alegere random a unui *i*. Deci, pentru fiecare *l* mutări de la algoritmul 1, îi corespunde *o singură* mutare la algoritmul 2. În schimb, la **3**, în funcție de anumite condiții, se realizează o minimizare a setului de date prin eliminarea unor *i*-uri, implicit valori verificate, deci, din nou, obținem un număr de pași mai mic decât la algoritmul 1.

Mai jos avem codul propriu-zis pentru cei 3 algoritmi:

```
""" ---------------     Algoritmii Propuși     ---------------"""
# Algoritmul 1: Dual Coordinate Descent Method (simplu)
def dual_descent(alpha_start, y, x, U, Q, w_start, D, tol):
    """
        Parametrii:
            alpha_start: alpha0, ales random (este același pentru toți cei
3 algoritmi, dar facem copie, ca să nu modificăm valoarea originală)
            y: {+1; -1} -> eticheta pentru un x; este un array de lungime l
            x: din R^n (x = x1, x2, ..., xn), dar avem un array de lungime
l de x
            U: din formula (4) din paper-ul original -> pt L1 este C
(paramaterul de penalizare), iar pt L2 e inf
            Q: (4) paper
            w_start: se obține din alpha_start (formulă în algoritm) (este
același pentru toți cei 3 algoritmi, dar facem copie, ca să nu modificăm
valoarea originală)
            D: (4) paper -> pt L1 este 0; pt L2 este 1/(2C)
            tol: toleranța (o eroare)
```

```
    """

    l = len(y)        # lungimea
    A = np.zeros(l)   # A[i] va deveni 1 când alpha[i] devine optimal și
ieșim din algoritm când toate val = 1
    k = 0             # nr pași / rata de convergență

    # Copiile
    w = w_start.copy()
    alpha = alpha_start.copy()

    # Repetăm până alpha devine optimal
    while True:
        for i in range(l):
            # a)
            G = y[i] * np.vdot(w, x[i]) - 1 + D[i][i] * alpha[i]

            # b)
            if abs(alpha[i]) < tol:
                PG = min(G, 0)              # PG = projected gradient
            elif U != float("inf") and abs(alpha[i] - U) < tol:
                PG = max(G, 0)
            else:
                PG = G

            # c)
            if abs(PG) > tol:
                alpha_aux = alpha[i]
                alpha[i] = min(max(alpha[i] - G/(Q[i][i] + D[i][i]), 0), U)
                w += (alpha[i] - alpha_aux) * y[i] * x[i]
                k += 1 # numărăm în câți pași se ajunge la un alpha optimal
            else:
                A[i] = 1  # altfel alpha[i] este optimal

        # verificăm dacă alpha este optimal (nu mai sunt valori de 0)
        if A.all() == 1:
            break

    return w, k


# Algoritmul 2: Dual Coordinate Descent Method with Randomly Selecting One
Instance at a Time
def dual_descent_rand(alpha_start, y, x, U, Q, w_start, D, tol):
    l = len(y)        # lungimea
    A = np.zeros(l)   # A[i] va deveni 1 când alpha[i] devine optimal și
ieșim din algoritm când toate val = 1
    k = 0             # nr pași / rata de convergență

    # Copiile
    w = w_start.copy()
    alpha = alpha_start.copy()

    # Repetăm până alpha devine optimal
    while True:
```

```
        # nu mai avem acel for (alegem random o valoare) -> reducem masiv
nr de operații și timpul
        i = random.randint(0, l - 1)

        # a)
        G = y[i] * np.vdot(w, x[i]) - 1 + D[i][i] * alpha[i]

        # b)
        if abs(alpha[i]) < tol:
            PG = min(G, 0)
        elif U != float("inf") and abs(alpha[i] - U) < tol:
            PG = max(G, 0)
        else:
            PG = G

        # c)
        if abs(PG) > tol:
            alpha_aux = alpha[i]
            alpha[i] = min(max(alpha[i] - G/(Q[i][i] + D[i][i]), 0), U)
            w += (alpha[i] - alpha_aux) * y[i] * x[i]
            k += 1
        else:
            A[i] = 1  # altfel alpha[i] este optimal

        # verificăm dacă alpha este optimal (nu mai sunt valori de 0)
        if A.all() == 1:
            break

    return w, k


# Algoritmul 3: Dual Coordinate Descent Method with Shrinking
def dual_descent_shrinking(alpha_start, y, x, U, Q, w_start, D, eps, tol):
    """
        Parametrul Suplimentar:
            eps: epsilon
    """

    l = len(y)
    M_ = float("inf")
    m_ = -1 * float("inf")
    A = list(range(l))
    A_vechi = []        # ca să verifici că s-a modificat lista originală
    k = 0

    alpha = alpha_start.copy()
    w = w_start.copy()

    while True:
        # 1.
        M = -1 * float("inf")
        m = float("inf")

        # 2.
        for i in A:
            # a)
```

```
            G = y[i] * np.vdot(w, x[i]) - 1 + D[i][i] * alpha[i]

            # b)
            PG = 0
            if abs(alpha[i]) < tol:
                if G > M_:
                    A.remove(i)
                    continue
                elif G < 0:
                    PG = G
            elif abs(alpha[i] - U) < tol:
                if G < m_:
                    A.remove(i)
                    continue
                elif G > 0:
                    PG = G
            else:
                PG = G

            # c)
            M = max(M, PG)
            m = min(m, PG)

            # d)
            if abs(PG) > tol:
                alpha_aux = alpha[i]
                alpha[i] = min(max(alpha[i] - G/(Q[i][i] + D[i][i]), 0), U)
                w += (alpha[i] - alpha_aux) * y[i] * x[i]
                k += 1

        # 3.
        if M - m < eps:
            if A_vechi == A:
                break                           # converge
            A_vechi = A                         # aici suntem pe linia else
            if A == list(range(l)):
                break
            else:
                A = list(range(l))

        # 4.
        if M <= 0:
            M_ = float("inf")
        else:
            M_ = M

        # 5.
        if m >= 0:
            m_ = -1 * float('inf')
        else:
            m_ = m

    return w, k
```

## *4.2. Eficiența și Rata de Convergență*

În urma testelor realizate pe un mini-set luat din dataset-ul a9a, putem observa următoarele:

```
----------  Loss L1 SVM  ------------
Algoritmul 1, Dual Coordinate Descent
W: [ 0.07195426  0.16692827  0.10507506  0.11397068  0.09549648 -0.23430028
 -0.12825977  0.13314251 -0.16159902 -0.10100157  0.01620227 -0.11537193
  0.26539229  0.02880559]
Numarul de pasi: 48039
Timpul: 1.0659170150756836 s
-------------------------------------
Algoritmul 2, Dual Coordinate Descent, Rand
W: [ 0.32064342  0.13874439 -0.0311603   1.78362324  0.70414243 -0.05354963
 -0.93085614  0.31605268 -0.07256175  0.04587627  0.23497554  0.11625914
  0.68494654 -1.58091343]
Numarul de pasi: 316
Timpul: 0.010478496551513672 s
-------------------------------------
Algoritmul 3, Dual Coordinate Descent with Shrinking
W: [ 0.05075328  0.12231776  0.10353525  0.08492914  0.14143643 -0.24795398
 -0.12530021  0.06230737 -0.1222208  -0.06987135  0.02804882 -0.09742816
  0.25949374  0.00086464]
Numarul de pasi: 457046
Timpul: 7.3384850025177 s
-------------------------------------
```

```
------------------  Loss L2-SVM  ------------------
Algoritmul 1: Dual Coordinate Descent (Simplu)
W: [ 0.1058457   0.11910427  0.08491266  0.00257656  0.18305769 -0.33548564
 -0.10384292  0.19875472 -0.12091071 -0.12298284  0.02161003 -0.20631644
  0.34344863  0.00438098]
Numarul de pasi: 8147233
Timpul: 110.44797849655151 s
-------------------------------------
Algoritmul 2: Dual Coordinate Descent Alegeri Random
W: [  2.73962328   1.72213523   0.17843179  32.98965405  10.05839152
 -10.79533523  -5.89988059   2.46854485  -8.19560327   0.8979767
  -0.7365697   -1.58601886  12.42248042 -10.73711974]
Numarul de pasi: 1190
Timpul: 0.023936033248901367 s
-------------------------------------
Algoritmul 3: Dual Coordinate Descent with Shrinking
W: [ 0.10584594  0.11910414  0.08491234  0.00257666  0.18305763 -0.33548556
 -0.10384216  0.19875392 -0.12091054 -0.12298206  0.02160962 -0.2063157
  0.34344785  0.00438073]
Numarul de pasi: 12780717
Timpul: 150.3523485660553 s
-------------------------------------
```

Presupunerile din analiza noastră anterioară se confirmă: remarcăm că atât algoritmul 2, cât și 3 au un număr de pași mai mic decât algoritmul 1.

## *4.3. Compararea Algoritmilor și Concluzii*

Din punct de vedere al ratei de convergență, se remarcă faptul că algoritmul 2 este cel mai eficient, urmat de algoritmul 3 și de 1. Ca timp de execuție, 2 continuă să își mențină poziția fruntașă, dar este urmat de algoritmul 1 și, apoi, de 3. Acest lucru se datorează operațiilor suplimentare, care ridică complexitatea algoritmului.

Ca direcție de îmbunătățit și testat, am putea să încercăm să „îmbinăm" algoritmii 2 și 3 pentru a putea observa dacă rezultatele obținute vor fi mai bune, în acest caz.

În concluzie, SVM este unul dintre cele mai populare instrumente folosite pentru clasificarea datelor, cu întrebuințări în viața de zi cu zi, de aceea este foarte important să găsim un algoritm care să aibă atât un timp scurt de execuție, dar și o acuratețe ridicată, mai ales pentru cazurile în care avem dataset-uri mari. Proiectul de față a prezentat un rezumat al lucrării scrise de Hsieh et al. și a încercat să reproducă experimentele acestora, la o scară mai redusă, pentru a ne însuși aceste cunoștințe.

# 5. Bibliografie


1. Deisenroth M-P, Faisal A-A, Ong C-S. *MATHEMATICS for MACHINE LEARNING*. https://mml-book.github.io/book/mml-book.pdf
2. Fan R-E, Chang K-W, Hsieh C-J, Wang X-R, Lin C-J. LIBLINEAR: A Library for Large Linear Classification. *Journal of Machine Learning Research*. 2008;9:1871-1874. https://www.csie.ntu.edu.tw/~cjlin/papers/liblinear.pdf
3. Hsieh C-J, Chang K-W, Lin C-J, Keerthi S, Sundararajan S. *A Dual Coordinate Descent Method for Large-Scale Linear SVM*. https://www.csie.ntu.edu.tw/~cjlin/papers/cddual.pdf
4. LIBLINEAR -- A Library for Large Linear Classification. Ntu.edu.tw. Published 2021. Accessed May 7, 2021. https://www.csie.ntu.edu.tw/~cjlin/liblinear/
5. Ihler A. Support Vector Machines (1): Linear SVMs, primal form. *YouTube*. Published online January 25, 2015. Accessed May 19, 2021. https://www.youtube.com/watch?v=IOetFPgsMUc&t=580s
6. Ihler A. Support Vector Machines (2): Dual & soft-margin forms. *YouTube*. Published online January 25, 2015. Accessed May 19, 2021. https://www.youtube.com/watch?v=1aQLEzeGJC8
7. SVMs. What is the influence of C in SVMs with linear kernel? Cross Validated. Published June 23, 2012. Accessed May 19, 2021. https://stats.stackexchange.com/questions/31066/what-is-the-influence-of-c-in-svms-with-linear-kernel?fbclid=IwAR1LhrsBYEan-L3CQWgKWXW7nTM4-4IAxwNiXfXzQQzvOm7JOXfDnF6lZs#:~:text=The%20C%20parameter%20tells%20the,the%20training%20points%20classified%20correctly
8. wjiang16. wjiang16/SVM-SGD-DCD. GitHub. Published January 14, 2018. Accessed May 19, 2021. https://github.com/wjiang16/SVM-SGD-DCD/blob/master/Dual_CD_SVM.py
9. *An Idiot's Guide to Support Vector Machines (SVMs)*. http://web.mit.edu/6.034/wwwbob/svm-notes-long-08.pdf
10. Shuzhan Fan. Understanding the mathematics behind Support Vector Machines. Shuzhan Fan. Published May 7, 2018. Accessed May 19, 2021. https://shuzhanfan.github.io/2018/05/understanding-mathematics-behind-support-vector-machines/
11. LIBSVM Data: Classification (Binary Class). Ntu.edu.tw. Published 2014. Accessed May 19, 2021. https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html#a9a